Selection Expressions for Procedural Modeling

SELEX Specification

1 Introduction

This document describes the syntax and semantics of the modeling language SELEX. The lexical and syntactic structure are given in Extended Backus-Naur Form (EBNF) aiming at a precise description (refer to [Wirth 1996]), while the semantics are explained using natural language. Given a program, lexical analysis separates strings (e.g. "a=2") as tokens (e.g. "a", "=", "2"), while syntax analysis operates on the stream of tokens (e.g. integer) and outputs a syntax tree. For the high-level concepts in the syntax analysis, more semantic details are elaborated on, in order to reveal the details of semantics and implementation.

We employ static type checking as the type system in SELEX. This means that the type of each symbol is determined during the parsing process. For example, command "a=1.2" will implicitly determine that variable "a" has type float.

The parser for SELEX is implemented in Python using the pyparsing library, which outputs a syntax tree. The data model and execution model in SELEX are implemented in C++. During runtime, we traverse the syntax tree, execute the commands evolving a 3D model, and output a final 3D model.

1.1 Notation

There are several variants of the EBNF notation. In our description, we adopt the conventions listed in Table 1.

Usage	definition	concatenation	alternation	optional
Notation	=	SPACE	|	[...]
	one or more	zero or more	grouping	terminal string
	+	*	(...)	"..."
	special sequence	escape character		
	?...?	\		

Table 1: *EBNF notation used in this document.*

2 Lexical analysis

The task of lexical analysis is to divide the strings of the input into a list of tokens.

2.1 Line structure

A SELEX program is composed of a list of commands, which are separated by a newline. However, a command can also span multiple lines containing newline characters.

Comments start with the character "#" and end at the end of the line. They will be ignored by the parser. However, the character "#" does not start a comment if it is part of a string. In this paper, as well as in our language, we use the term character to refer to any type of ASCII character. We use the term letter to refer to alphabetic characters.

```
comment = "#" (char)*
char = ?any ASCII character?
letter = ?alphabetic characters a-z and A-Z?
```

2.2 Identifiers and keywords

Identifiers can be used to identify a variable, an attribute, or a function name, e.g. "facHeight" could be the name of a variable that stores the facade height. SELEX is case-sensitive and an identifier is defined as follows:

```
identifier = letter (letter | digit | "_")*
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Keywords are reserved by the language, and cannot be redefined by the user:

```
"child" "descedant" "parent" "root" "self" "neighbor"
"label" "type" "rowIdx" "colIdx" "rowLabel" "colLabel"
"last" "rowLast" "colLast" "groupRows" "groupCols"
"groupRegions" "if" "randomSelect" "eval"
```

2.3 Literals

Numeric literals define numbers (floating point or integer):

```
float = ["-" | "+"] digit+ "." digit+
integer = ["-" | "+"] digit+
number = float | integer
```

Boolean literals can be either false (0) or true (1):

```
Boolean = "0" | "1"
```

In SELEX, the automatic type-casting between numeric literals and Boolean literals is allowed as in C/C++.

String literals define a sequence of characters as follows:

```
string = "\"" (char)* "\""
```

2.4 Operators and delimiters

Operators can be the following:

```
"+" "-" "*" "/" "in" "contains" "!" "&&" "||"
">" "<" "==" ">=" "<=" "!="
```

Delimiters are the following:

```
"{" "}" "[" "]" "(" ")" "<" ">"
"=" "," ":" ";" "->"
```

Whitespace characters are special characters used to separate tokens.

```
whitespace = "\n" | " " | "\t"
```

3 Syntax and semantic analysis

3.1 Data model

We use object as the abstraction of different types of data. Every object has a type and a pointer to its value. The value pointer stores the address of the value in memory. In SELEX, we support the following types: Boolean, float, integer, string, list, pair, shape, and construction-line. A list is a list of other types of objects and we also support lists of lists. A pair consists of two objects. The first object should be comparable, and can be a number, a string, or a Boolean value. The second object can be any kind of object. A construction-line is a special type of object used to create virtual shapes.

In the language, a **list** is defined as follows.

```
list = "(" expression ("," expression)* ")"
```

3.2 Selection-expressions

A **selection-expression** selects a list of shapes from the shape tree using selectors interleaved with the operator "/". Each selector takes a list of shapes as input and returns a list of shapes. The implicit input to the first selector is a list containing the root node of the shape tree. The operator "/" takes a list of shapes as input and executes the remaining commands for each shape in the list.

Selectors are grouped in selector sequences ("selectorSeq") that consist of specialized selectors that can have three different types: topology-selector (e.g. child, descendant), attribute selector (e.g. "[label=="window"]") and group selector (e.g. "[::groupRows()]"). The selectors cannot be arbitrarily mixed within a sequence and they need to occur in the given order.

A topology selector takes a list containing a single shape as input, and outputs a list of shapes with the specified topology relation to the input shape. An attribute selector takes a list of shapes as input, and returns a list of shapes whose attributes satisfy some conditions. A group selector takes a list of shapes as input, and applies grouping operations to return a list of combined shapes. A group selector only operates on virtual shapes and regroups subregions, e.g. combines cells of a virtual shape into floors. If a selection-expression is empty, it returns the input.

```
selectionExpression = "<" [selectorSeq ("/" selectorSeq)*] ">"
selectorSeq = [topoSelector] [attrSelector | groupSelector]*
topoSelector = funcCall
attrSelector = "[" boolExpr "]"
groupSelector = "[" "::" funcCall "]"
```

where "boolExpr" encodes the set operation and comparison operation and "funcCall" calls a function as described in Sec. 4. Note that the given syntax is not very restrictive. However, the semantics only accepts certain type of function calls and Boolean expressions to be used.

3.3 Variable and function

A **variable** is initialized by an expression. Afterwards, the value of the variable can be referred to by the identifier:

```
assignment = identifier "=" expression
```

Function is called by the command:

```
funcCall = identifier "(" argList ")"
argList = [expression ("," expression)*]
```

Here expression acts as an argument for a function, e.g. "0.5+0.1" in "toShapeX(0.5+0.1)". Currently, we do not allow the definition of new functions. Only functions from our given function library can be called.

3.4 Expression

An **expression** evaluates to a value and it combines variables (identified by an identifier), functions, arithmetic operations, Boolean operations, and set operations. The arithmetic operation operates on two numeric value, and returns the result. A Boolean operation takes one or two values as input, and returns a Boolean result. A set operation tests if an object is contained in a list. The detailed definition is given in the following EBNF. Note that the arithmetic operation, Boolean operation, and set operations are defined recursively to enable nested expressions. For example, a Boolean operation can operate on a value returned by an arithmetic operation, e.g. "1+4==5".

```
expression = identifier | funcCall | boolExpr

boolExpr = andExpr ["||" andExpr]
andExpr = notExpr ["&&" notExpr]
notExpr = ("!" boolOperand) | boolOperand
boolOperand = setExpr | cmpExpr | ("(" setExpr ")")

setExpr = (expression "in" list) | (list "contains" expression)

cmpExpr = arithExpr [cmpOP arithExpr]
cmpOp = "==" | "!=" | ">=" | "<=" | ">" | "<"

arithExpr = multiplyExpr [("+"|"-") multiplyExpr]
multiplyExpr = arithOperand [("*"|"/") arithOperand]
arithOperand = expression | ("(" arithExpr ")")
```

Table 2: *List of expression operators.*

Operator	Effect
x, /, +, -	algebra operations
!, &&, ||	Boolean operations
==, !=, >, <, >=, <=	comparison operations
in, contains	set operation

3.5 Execution model

A SELEX program is a collection of commands. Each command can be a rule, an assignment, or a return statement. Each command is interpreted and executed one by one. An exit statement will exit the program, and any command after it will not be executed:

```
program = (command)*
command = rule | assignment | exit
exit = "exit;"
```

A **rule** has two components, i.e. selection-expression and actions. Actions are a list of functions that act on a shape selected by a selection-expression:

```
rule = "{" selectionExpression "->" actions "}"
actions = (funcCall ";")+
```

The implemented actions are listed in Sec. 4.2

Naming and binding determines what an identifier refers to in SELEX. In our implementation, it can be a variable (e.g. "fac-

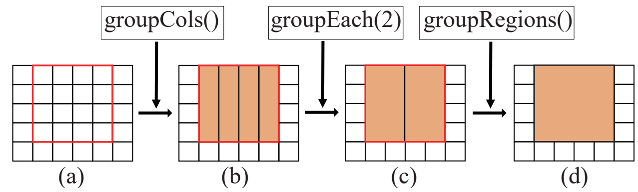


Figure 1: *Given the red selected region of (a), command* "groupCols()" *groups the cells into columns to create a list of virtual shapes (shown in orange in (b)). Then command* "groupEach(2)" *groups adjacent columns to yield a list of two regions shown in (c). At last, command* "groupRegions()" *combines the virtual shapes into a single region shown in (d).*

Size"), a function (e.g. "numCols" in "numCols()") or an attribute (e.g. "label" in "[label == "facade"]") depending on its context.

A variable is created by an assignment statement, and stored as a name and object pair. When we refer to a variable, the object will be queried and returned.

An attribute in SELEX can be a built-in attribute or a dynamic attribute. A built-in attribute is created by SELEX, and a dynamic attribute is created by a user when they call some specific commands, e.g. "setAttrib(...)".

A function is called with its arguments, and returns an object. The function name will be queried and arguments are matched before the execution of the function.

We currently do not support local scopes. A variable can be used anywhere after it has been defined. However, an attribute can only be used inside a selection-expression.

4 The built-in function library

The following types of functions are supported:

- selection functions which help select shapes
- shape functions which refine the shape hierarchy
- utility functions for coordinate conversion and the query of information
- constraint functions which specify necessary constraints on shapes
- auxilliary math functions

For an easier understanding of the text, we do not describe the built-in function library in EBNF form.

4.1 Selection functions

There are three types of selection functions: the topology selection functions, attribute testing functions, and grouping functions. These functions are used in the corresponding selectors.

Topology functions select shapes based on their topological relation with the input shape, which can be the children, the parent, the descendants, the root shape and the left or right neighbors. The corresponding functions are listed below.

```
child();
descendant();
parent();
root();
neighbor(["left"], ["right"]);
```

Attribute testing functions test the attributes of the shapes in the input list and produce a list of selected shapes which pass the test. A shape passes an attribute test if it returns true on the test.

The function "isEmpty()" tests if a shape does not have any construction shapes inside. Another version "isEmpty(selection1)" checks if there is any selected shapes in the selection "selection1".

Function "pattern(regex, pat)" checks if the pattern character of "regex" at the index position of a shape matches "pat". For example, "pattern("(AB)*", "A")" tests if an input shape is at an odd index position, and "pattern("A(B)*A", "A")" tests if an input shape is at the first or last position of an input list. Also, more complex examples are possible and meaningful, e.g. "pattern("AC(ACCA)*CA", "A")", but regular expressions have inherent ambiguities when multiple repetitions are used. For example, for the case "pattern("A*B*A*"", "A")", we try to keep an equal amount of repetitions. Nested repetitions are also ambiguous and currently not supported.

Function "isEven()" and "isOdd()" are special cases of the command "pattern(regex, pat)", which check if a shape has an even or odd index in a list of selected shapes.

Many common attribute tests are formulated as a Boolean expression. For example, "label == "win"" tests if a shape has a label "win". Except for the built-in attributes, we use the index as an attribute to facilitate the test based on the index of an input shape in a list, or a grid. For example, the 5th shape can be selected by "idx==5". "idx" is the index of a shape in a list. "rowIdx" and "colIdx" is the topological position of a cell with respective to the region spanned by input virtual shapes. For example, "rowIdx==1 && colIdx==1" specifies the left bottom cell of a region spanning by given virtual shapes.

```
isEmpty([selection1]);
pattern(regex, pat);
isEven();
isOdd();
```

Grouping functions operate on a list of shapes and return a new list of shapes. Generally, grouping functions are used to restructure subregions of a virtual shape (grid) in different ways.

Function "groupRows()" and "groupCols()" merge adjacent virtual shapes (i.e. cells) with the same row or column index. Function "groupRegions()" merges all adjacent virtual shapes, which form one or multiple rectangular regions. We show an example in Fig. 1. Function "groupEach(n)" merges every n adjacent virtual shapes. Function "groupPair()" generates all possible pairings of two subsequent shapes. For example, given "ABCD" it will return "AB", "BC", and "CD". Function "cells()" decomposes a virtual shape as a list of virtual shapes with one cell. Function "sortBy(d, pos, order=1)" sorts the selected shapes by relative position "pos" in the dimension "d" with the increasing (order=1) or decreasing (order=0) order.

```
groupRows();
groupCols();
groupRegions();
groupEach(n);
groupPair();
cells();
sortBy(dim, pos, order=1);
```

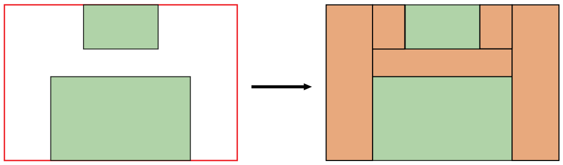


Figure 2: *Left: A construction shape shown with a red boundary contains two green construction shapes. Right: Executing command* "coverShape()" *creates several orange shapes as children of the red shape.*

4.2 Shape functions

Shape functions are used on the right hand side of selection-rules. Each shape function has an implicitly defined input shape. Typically, the input shape is one element of a list of shapes that is returned by a the selection-expression used on the left hand side of the selection-rule. A shape function returns a status flag, i.e. false for failure and true for success. Note that not all shape functions make use of the input shape.

Function "addShape" adds a 2D construction shape to another 2D construction shape. Parameter "la" specifies the label of the new shape, parameters "cx, cy, w, h" specify the center point and size, parameter "offset" the relative depth with respect to its parent, and parameter "visible" controls the visibility of the shape. The last two parameters are optional.

There are different versions of "addShape". Function "add2ProjectedLeafShape" adds a construction shape to the leaf construction shape of the input shape, which contains the projection of the added shape. These versions also exist for other commands, such as "attachShape".

```
addShape(la, cx, cy, w, h [, offset=0.0, [visible=1]]);
add2ProjectedLeafShape(la, cx, cy, w, h [, offset=0.0,
       [visible=1]]);
```

Function "attachShape" attaches a construction shape to another construction shape. The parameters are similar to function "addShape" to specify the 2D size and location. In addition the parameters "near-offset, far-offset" specify the minimal and maximal depth values relative to the input shape to yield a 3D attached shape.

```
attachShape(la, cx, cy, w, h, near-offset, far-offset);e
```

Function "connectShape" connects any two descendant construction shapes of an input shape, which are adjacent in the 2D projected XY plane and have different depth values in the coordinate frame of their sharing parent.

```
connectShape();
```

Function "coverShape" adds a set of shapes to an input shape so that each descendant shape will be partitioned by its children in the 2D projected plane of the local coordinate frame (see Fig. 2 for an example).

```
coverShape();
```

Function "copyShape" copies all descendants of the shape in selection "selection1" and maps their coordinates to the input shape. Parameter "transformation" is a list of transformation commands (e.g.

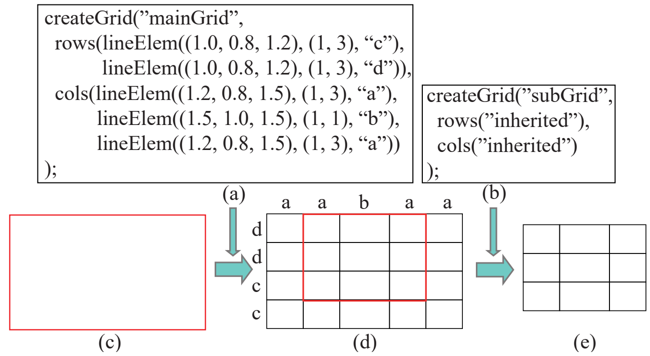


Figure 3: *Starting from a region shown in (c), the actions in (a) create a virtual shape (grid) shown in (d). The row ("ddcc") and column labels ("aabaa") are specified by the actions. We can add a sub-grid in the red region in (d) by executing the actions in (b). The resulting grid, shown in (e), re-uses the cells of the parent grid.*

"scale:(1,-1,1)" mirroring a shape along the y-direction). All shapes are transformed and added as children to the input shape.

```
copyShape(selection1, transformation);
```

Function "polygon" is constructed by a set of 2D points where each point is specified by a two element list, e.g. "(x1,y1), (x2,y2), ...":

```
polygon(point1, point2, ...);
```

Function "addVolume" adds a volume with label "la" as child of the input shape by extruding a polygon "polygon" with height "h". The label for each face of the volume is specified in the list "labels".

```
addVolume(la, polygon, h, labels);
```

Function "lineElem" has three parameters: "spacing" is a three element list specifying the preferred, minimal, and maximal distance from the previous construction line or the boundary. The repetition "rep" is a two-element list specifying the minimal and maximal number of repetitions. The label "la" is given as a string. For example, the command "lineElem((4.0, 3.5, 5.5), (1,1), "groundFloor")" will add a construction line with spacing as close as possible to 4.0 while remaining in the interval $[3.5, 5.5]$. The construction line cannot be repeated and it is labeled with "groundFloor".

```
lineElem(spacing, rep, la);
```

Function "Group" can be used to group several construction lines in order to repeat the whole group. For example, the command "group(A, B)" can be used to create a repetition of the form "(AB)*".

```
group(constructionLine1, constructionLine2, ...);
```

Function "createGrid" adds a virtual shape. The function has three parameters: the label "la", a list of construction lines in the horizontal direction (i.e. "rs") and a list of construction lines in the vertical direction (i.e. "cs"). The corresponding functions "rows" and "cols" return such lists of contruction lines. If "rows" and "cols" are called with the keyword "inherited" the list of construction lines are copied from the parent. Fig. 3 shows an example.

```
createGrid(la, rs, cs);
```

```
rows(constructionLine1, constructionLine2, ...);
rows("inherited");
cols(constructionLine1, constructionLine2, ...);
cols("inherited");
```

Function "SetAttrib" sets or adds an attribute with name "name" and value "val" to the input shape, and the name can be used to query this attribute.

```
setAttrib(name, val);
```

Function "exchange" exchanges a selected row (column) of an input shape with the row (column) of another selection "selection1".

```
exchange(selection1);
```

Function "transform" combines rotations, translations, and scaling and returns the transformation information as a list. The individual transformations can be specified by the corresponding functions "rotate", "translate", and "scale".

```
transform([scale [,translation [,rotation]]]);
rotate(xrot, yrot, zrot);
translate(xpos, ypos, zpos);
scale(xs, ys, zs);
```

Function "include" will load a specified asset "la" and transform it with the transformation information specified as the list "trans". This list is generated by the function "transform" described above.

```
include(la, trans);
```

Function "shareCorner" automatically generates a completion of a facade part according to its adjacent facades.

```
shareCorner();
```

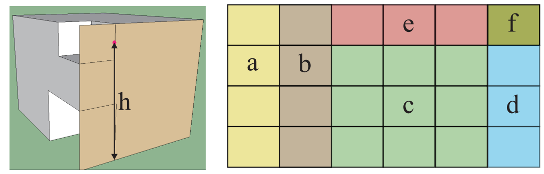
Function "finalRoof" is a specialized command to generate roof shapes.

```
finalRoof(h1, h2, ...);
```

4.3 Other utility functions

A coodinate conversion function converts positions between different coordinate systems. The parameter "val1" specifies a position and the parameter "selector1" is a list containing one shape that is typically generated by a selection-expression. If no selection-expression is specified, the input shape becomes the reference shape. Function "toParent?" converts the value "val1" to the coordinate system of the reference shape's parent. Function "toShape?" converts the value "val1"to the coordinate frame of the reference shape. Function "toLocal?" converts a position in the parent coordinate frame to the coordinate frame of the reference shape. We use normalized values for these mappings. We map each dimension of a shape to a normalized value in the interval [0, 1]. For example, the center position in the x-dimension corresponds to the normalized value 0.5.

```
toParentX(val1 [, selector1])
toParentY(val1 [, selector1])
toShapeX(val1 [, selector1])
toShapeY(val1 [, selector1])
toLocalX(val1 [, selector1])
toLocalY(val1 [, selector1])
```



(a) (b)

Figure 4: *Two query examples. (a) For the input facade shown in brown, command* "queryCorner" *queries the x or y coordinate of the corner point shared with the white facade specified by a selection-expression in its left or right side with height* h*. (b) The functions* "numRows, numCols, rowLast, colLast, rowRange, colRange" *work on an input list of virtual shapes that form a rectangular region. For example, for the input list of virtual shapes (c, d), which are adjacent and can be merged into a larger region, functions* "numRows"' *and* "numCols" *will yield 3 and 4 respectively.*

The function "queryCorner" is used to compute the position of a corner. This is typically useful if corners were cut out of a facade. The corner position of the input shape is computed at side "side" and height "h". The corner is with respect to the first shape specified by "selector" projected to the input shape. An example is shown in Fig. 4a.

```
queryCorner(selector, side, h, whichDim);
```

List query functions take a list of shapes as input, and return information about the input list.

The function "count()" returns the number of elements in the input list. The function "last(i)" returns the last i-th index of the input list. For example, if we have 5 shapes in the input list then the function "last(1)" returns the last index 5. The function "indexRange(idxBegin, idxEnd)" returns a list of indices from "idxBegin" to "idxEnd". Function "index(i)" returns the i-th shape in the input list.

We also provide functions to query virtual shapes. These queries can be based on rows or columns. The functions "numRows()" and "numCols()" return the number of rows and columns of the region spanned by the virtual shapes in the input list (see Fig. 4b). The functions "rowLast(i)" and "colLast(i)" return the last i-th index of rows and columns of all virtual shapes in the input list. The function "rowRange(idxBegin, idxEnd)" and "colRange(idxBegin, idxEnd)" return a list of shapes, which are inside the region spanned by rows (or columns) "idxBegin" to "idxEnd", and include both "idxBegin" and "idxEnd". We also allow the use of negative numbers that count from the last index backwards, e.g. -1 indicates the second to last index.

```
count();
last(i);
indexRange(idxBegin, idxEnd);
index(i);

numRows();
numCols();
rowLast(i);
colLast(i);
rowRange(idxBegin, idxEnd);
colRange(idxBegin, idxEnd);
```

4.4 Constraint functions

In general, it is difficult to specify the location of a shape in a stochastic grammar, since we cannot know exactly what shapes have been placed previously. Therefore, a user can use SELEX to specify a sequence of constraints and leave the precise shape placement to an optimization algorithm. It is possible to use the optimization in conjunction with the two functions "addShape" and "attachShape".

A sequence of constraints can be specified with the following command:

```
constrain(constraint1, constraint2, ...),
```

A constraint is given by a constraint function, which will take the input shape as an implicit parameter and output a constraint specification in list form. Each constraint specification includes variable names, variable weights, and the comparison operation. Supported constraints are elaborated on below.

The specified constraints may be compatible or not. To tackle potential conflicts in the constraints, we incrementally check the compatibility. If no conflict is detected, we just add the constraint to the constraint set. Incompatible constraints are dropped. That means, that constraints specified first implicity have a higher priority. At last, an optimizer will enforce the selected constraints to obtain optimal shape parameters.

The optimization uses a quadratic objective function with linear constraints. The optimization computes the optimal (final) lower left position (x^*, y^*) and optimal size (w^*, h^*) of a 2D shape. The objective function encodes that the final position and size should be close to the specification (x, y, w, h) in a least squares sense:

$$(x^* - x)^2 + (y^* - y)^2 + (w^* - w)^2 + (h^* - h)^2, \qquad (1)$$

Alignment can be specified between two shapes. The input shape and a reference shape specified by a shape label. We support the following types of alignment: "left", "right", "top", "bottom", "center-x", "center-y", "one2two-x", "one2two-y".

```
snap2(shapeLabel1, snapType1, shapeLabel2, snapType2, ...)
```

For example, the function "constrain(snap2("window1", "left"), snap2("window1", "center-x"))", specifies that the input shape should be left and center-x aligned with a shape labeled "window1".

Here we describe the formulation of alignments in more detail. Our alignment scheme has three steps. First, we detect the reference shapes that may align to the input shape. Then, we calculate the snapping position according to the alignment type. At last, we enforce the alignment between the current shape and the preferred position according to the alignment type using an optimization technique. In the following, we take bottom alignment as an example to describe the algorithm, and explain the special alginments "one2two-x", "one2two-y". The corresponding examples can be found in Fig. 5.

In the detection step, shapes that with the specified label are first selected. Alignment "left", "right", "top", "bottom", "center-x", "center-y" align one element to another as shown in Fig. 5(a). But alignment "one2two-x", "one2two-y" try to align one element to the bounding box of two nearby elements with the given label. Thus, the bounding boxes are returned as the selected shapes as shown in Fig. 5(b). Among the selected shapes, the final candidates are shapes that satisfy the specified alignments to input shape within a threshold (half of the width or height of the input shape in our experiments). For example, left alignment will test if the difference between left edges of the selected shape and the input shape is within the threshold. For alignment "one2two-x", "one2two-y", center alignment between the returned bounding box and input shape is tested.

In the second step, snapping position s_i is calculated. For example, left alignment will use the nearest edge of the selected shape with respect to the left edge of the input shape as illustrated in Fig. 5(a). For alignment "one2two-x", "one2two-y", the nearest horizontal or vertical center position relative to the horizontal or vertical center position of the input shape will be used, as illustrated in Fig. 5(b).

At last, alignment can be achieved by adding the alignment constraints to an optimization. Assuming we would like to align to a position s_i, the constraint is formulated as: $x^* + \alpha_i * w^* = s_i$, where x^*, w^* is the left position and width of a shape, and α_i equals to $-0.5, 0.0, 0.5$ for left, center-x, and right alignment, respectively.

If multiple alignments are specified within a snap2 function, one of these alignments should be enforced. For example, the function "constrain(snap2("window1", "left", "window1", "center-x"))", specifies that the input shape should be either left or center-x aligned with a shape labeled "window1".

Selecting one constraint from n equality constraints of a form $x^* + \alpha_i * w^* = s_i$ can be reformulated as a set of linear constraints as follows:

$$\begin{aligned} x^* + \alpha_i * w^* - s_i + M * b_i \geq 0, &\forall i \in [1, n], \\ x^* + \alpha_i * w^* - s_i - M * b_i \leq 0, &\forall i \in [1, n], \\ \sum_j^n b_j = n - 1&, \\ b_i \in \{0, 1\}, \forall i \in [1, n]&, \end{aligned} \qquad (2)$$

Local symmetry refers to the alignment to the center of the input shape, which is specified by

```
sym2region();
```

This can be formulated as $x^* + \alpha_i * w^* = s_c$,.

Distance to boundary constrains the minimal and maximal distance to the left, right, top, or bottom of a reference region. The reference region is either the input shape for "dist2region" or a parent shape of the input shape selected by its label for

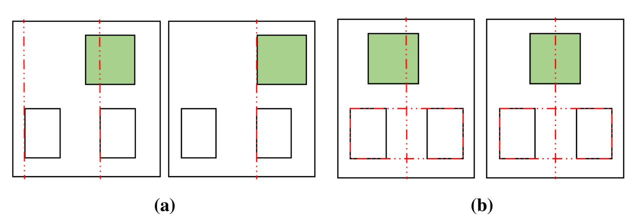


Figure 5: *Two example alignments. In each subfigure, the left side is derived without alignments, while the right side is derived with alignments. (a) Alignment "left" aligns the input shape in green to a reference shape in white. (b) Alignment "one2two-x" aligns the input shape in green to the center of the bounding box of two white reference shapes. The red dashed line denotes the snapping position, while the red bounding box marks the bounding box of two reference shapes.*

"dist2layout"). The auxilliary functions "dist2left", "dist2right", "dist2top", "dist2bottom" generate a constraint specification in list form.

```
dist2layout(label,[dist2func, ...]);
dist2region([dist2func, ...]);
dist2left(minDist, maxDist);
dist2right(minDist, maxDist);
dist2bottom(minDist, maxDist);
dist2top(minDist, maxDist);
```

For example, "dist2region(dist2bottom(0.0, 0.0))" means that the input shape should touch the bottom of the newly generated shape. The constraints can be specified as follows. We just use the command "dist2left(d_0, d_1)" as example:

$$\begin{aligned} x^* - 0.5 * w^* - d_0 \geq 0, \\ x^* - 0.5 * w^* - d_1 \leq 0. \end{aligned} \tag{3}$$

The specifications for "dist2right", "dist2bottom", "dist2top" are analogous.

Intersection avoidance prevents invalid intersections between the newly generated shape and other shapes. In SELEX, users can specify the valid intersections, then the other intersections will be invalid. The intersection command has the following syntax:

```
validIntersect(la1, la2, ...);
```

where "la1", "la2" are labels of shapes that can intersect with the current shape.

These intersection constraints are formulated as:

$$\begin{aligned} &min(x_1^* - 0.5 \cdot w_1^*, x_2^* - 0.5 \cdot w_2^*) \\ &\quad \leq max(x_1^* + 0.5 \cdot w_1^*, x_2^* + 0.5 \cdot w_2^*), \\ &min(y_1^* - 0.5 \cdot h_1^*, y_2^* - 0.5 \cdot h_2^*) \\ &\quad \leq max(y_1^* + 0.5 \cdot h_1^*, y_2^* + 0.5 \cdot h_2^*). \end{aligned} \tag{4}$$

By the big-M method, we can convert these two functions into linear form:

$$\begin{aligned} y_1^* + h_1^* - y_2^* - M * \delta_3 - M * \delta_2 - M * (1 - \delta_1) \leq 0, \\ y_2^* + h_2^* - y_2^* - M * (1 - \delta_3) - M * \delta_2 - M * (1 - \delta_1) \leq 0, \\ y_1^* + h_1^* - y_1^* - M * \delta_4 - M * (1 - \delta_2) - M * (1 - \delta_1) \leq 0, \\ y_2^* + h_2^* - y_1^* - M * (1 - \delta_4) - M * (1 - \delta_2) - M * (1 - \delta_1) \leq 0, \\ x_1^* + w_1^* - x_2^* - M * \delta_6 - M * \delta_5 - M * \delta_1 \leq 0, \\ x_2^* + w_2^* - x_2^* - M * (1 - \delta_6) - M * \delta_5 - M * \delta_1 \leq 0, \\ x_1^* + w_1^* - x_1^* - M * \delta_6 - M * (1 - \delta_5) - M * \delta_1 \leq 0, \\ x_2^* + w_2^* - x_1^* - M * (1 - \delta_6) - M * (1 - \delta_5) - M * \delta_1 \leq 0, \end{aligned} \tag{5}$$

where M is a big value (10000 in our implementation), and $\delta_i, \forall i \in [1, 6]$ are binary variables.

4.5 Math functions

We offer the following functions to obtain random numbers: Function "randint" generates a random integer within the interval [min, max], "rand" generates a random floating point within the interval [min, max], and "randSelect" selects a value from the given list of values with a uniform distribution.

```
randint(min, max);
rand(min, max);
randSelect(val1, val2, val3, ...);
```

4.6 Flow control and stochastic variations

Conditional rules are one necessary ingredient for specifying variations. In SELEX, conditional rules are implemented using a special function "if", which takes a Boolean expression "cond" as first input. Optional parameters "selectionExpression" and "funcCall" will be executed if the given condition "cond" evaluates to true.

```
if(cond [, selectionExpression | funcCall]);
```

In SELEX, we chose not to implement stochastic rules directly, but to rely on a combination of random variables and conditional rules. For example, we may want to add two kinds of windows randomly, which can be programmed as
"a = rand(0.0, 1.0);
if(a>0.5, addShape("win1", ...));
if(a<=0.5, addShape("win2", ...));"
Note that we abbreviated the parameters for "addShape" in the given example.

To allow for the execution of a rule encoded as a string, SELEX supplies an evaluator. This could also be used to introduce randomness if the string is generated during the execution of the SELEX program.

```
eval(string);
```

5 Modeling example

In Figs. 6-20, we give details about a modeling example shown in the paper.

References

WIRTH, N. 1996. Extended backus-naur form (ebnf). *ISO/IEC 14977*, 2996.

Figure 6: *(a) The starting shape is a rectangle. The rectangle is selected and extruded. In this series of images, we always show the state before a command is processed.*

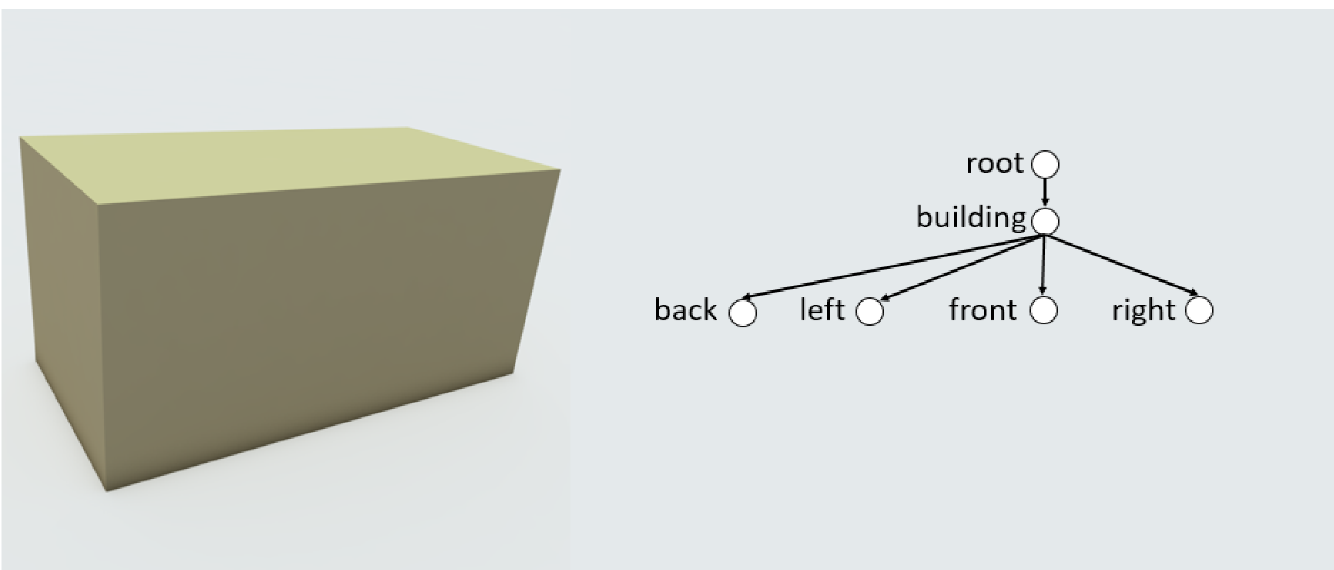


Figure 7: *(b) The building shape is selected and a grid is inserted as a virtual shape to split the building into floors. The floors are consistent across all building facades. This grid works similar to construction lines in technical drawing and does not actually split the building geometry. The corresponding selection-expression is* "<[label=="building"]>".

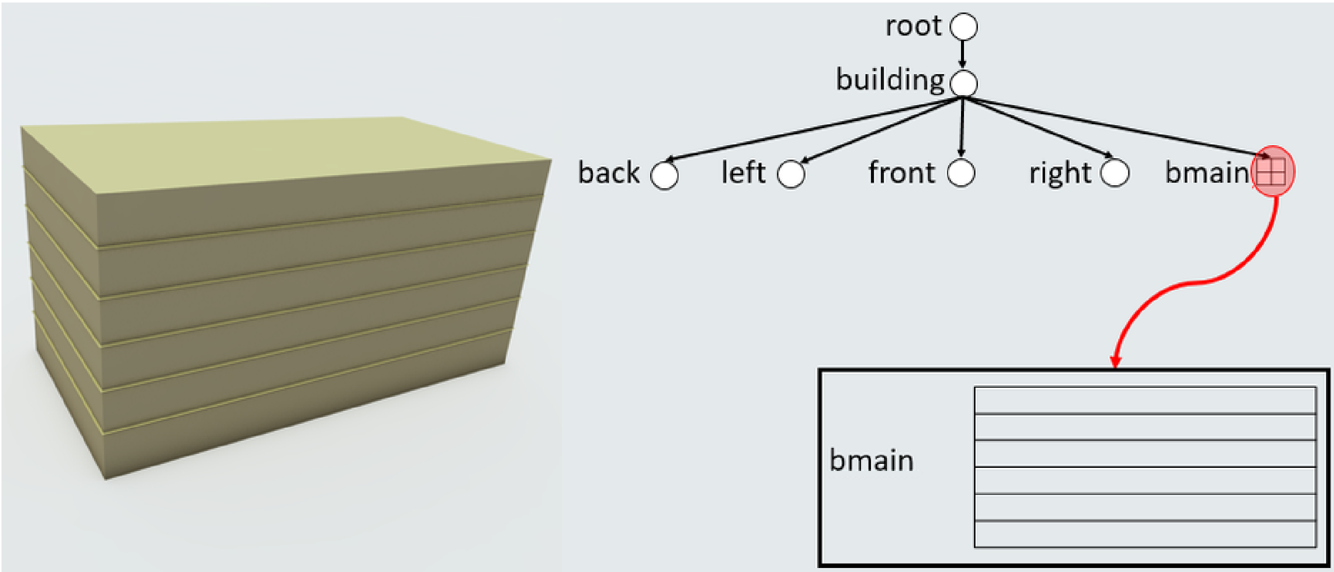


Figure 8: *(c) The front facade is selected and a grid named* "fmain" *is inserted. The corresponding selection-expression is* "<[label=="building"] /[label=="front"]>".

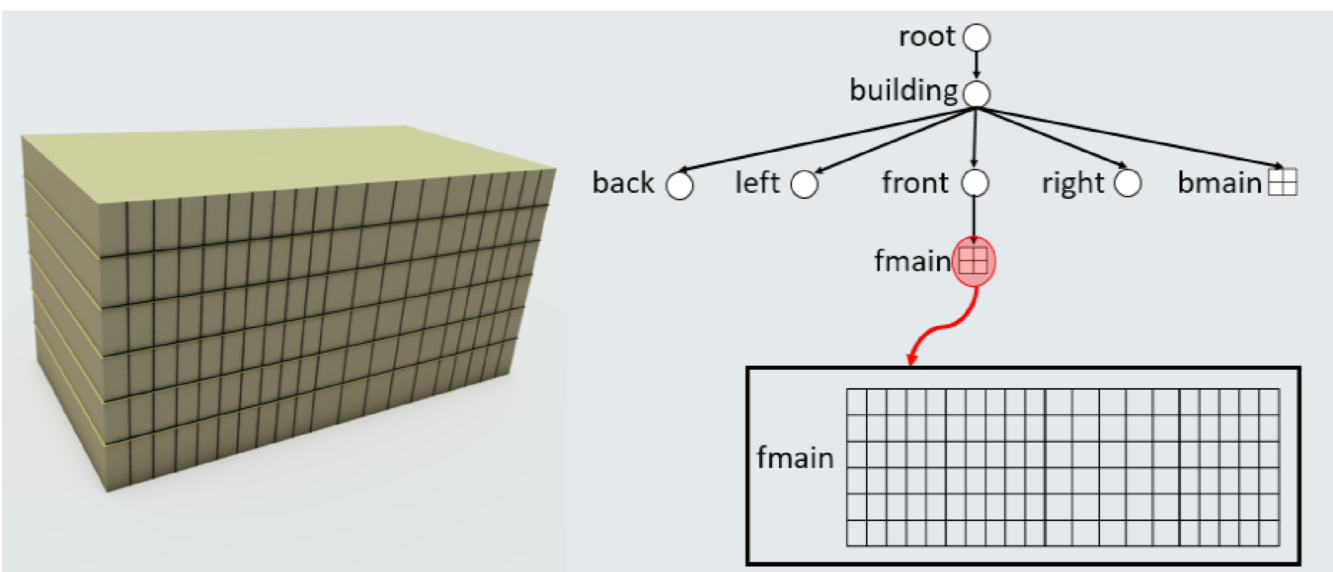


Figure 9: *(d) Each facade inherits the floor information and is split into a finer grid by specifying columns. The columns are labeled with* "colLeft", "colMidLeft", "colMidRight", "colRight".

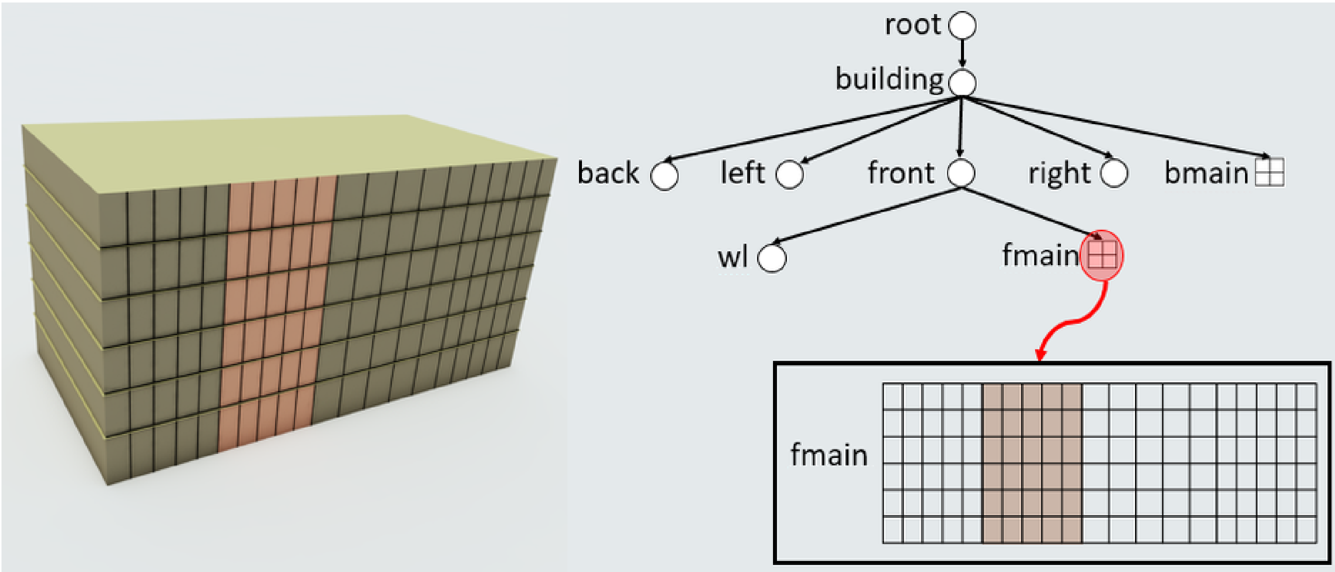


Figure 10: *(e) The grid named* "fmain" *is selected and a subgrid is selected, moved backwards, and inserted as construction shape* "wl". *The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="fmain"] /[type=="cell"] [colLabel=="colMidLeft"][::groupRegions()]>".

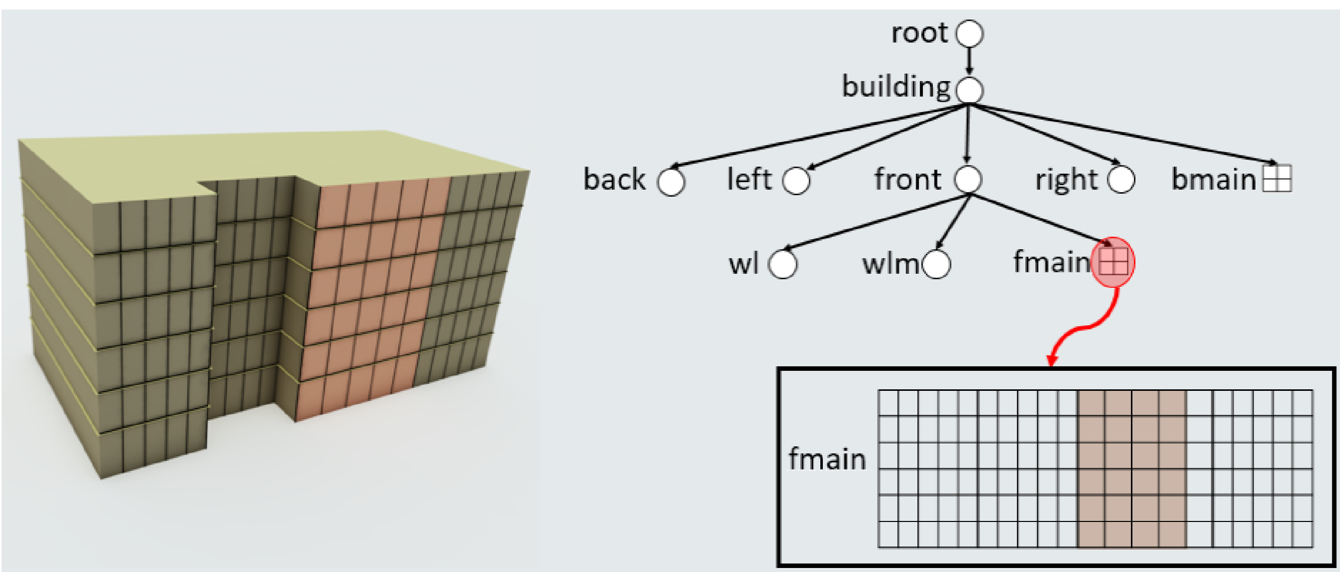


Figure 11: *(f) The grid named* "fmain" *is selected again and another subgrid is selected, moved forward and the corresponding geometry is inserted as child of front named* "wlm". *The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="fmain"] /[type=="cell"][colLabel=="colMidRight"][::groupRegions()]>".

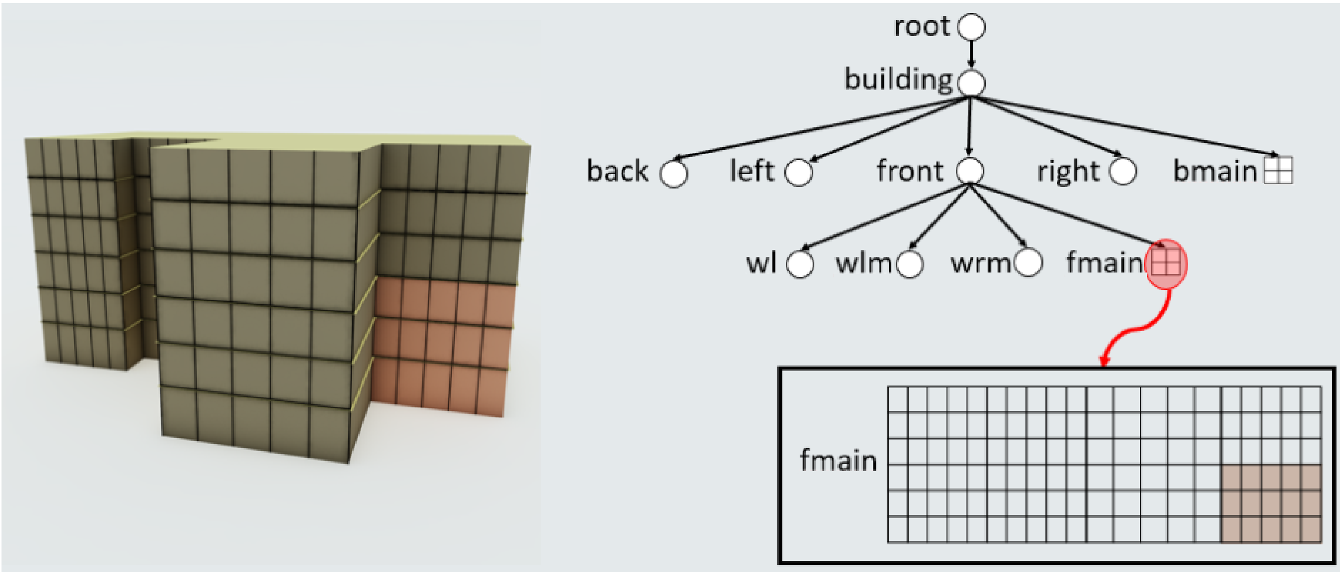


Figure 12: *(g) Another subgrid of* "fmain" *is selected, moved forward, and the corresponding geometry is inserted. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="fmain"] /[type=="cell"][colLabel=="colRight"][::groupRegions()]>".

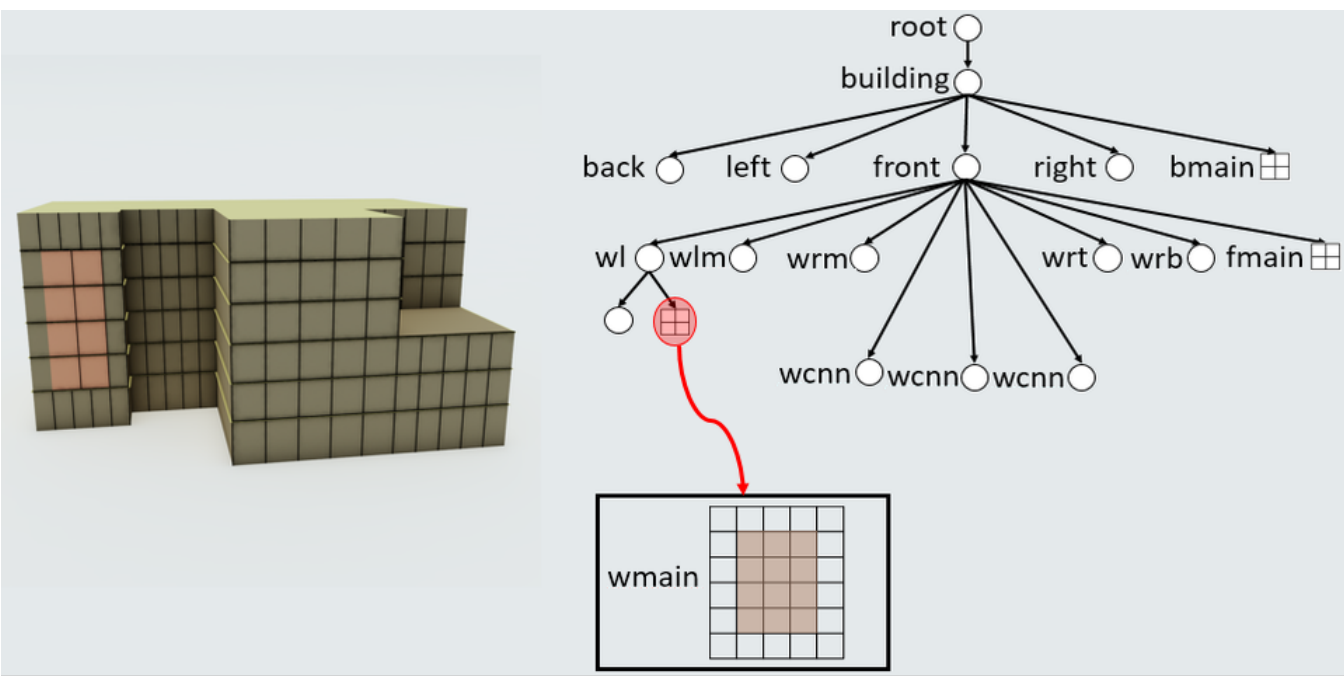


Figure 13: *(h) A subgrid of* "wl" *is selected in order to insert some larger windows. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="fmain"] /[type=="cell"][colLabel=="colMidLeft"][::groupRegions()] /[type=="cell"][rowIdx in rowRange(2,-2)][colIdx in colRange(2,-2)][::groupRegions()]>".

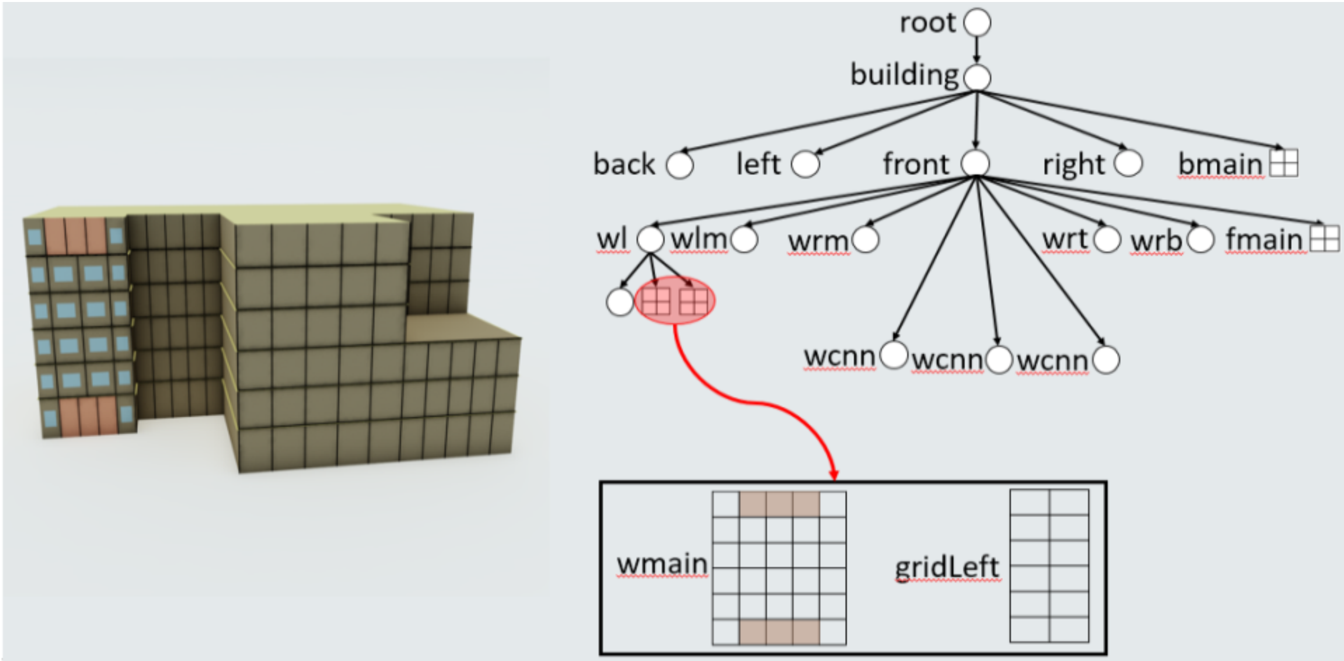


Figure 14: *(i) The selected subgrids will be used to insert a door in the first floor and a large window on the top floor. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="wl"] /[label=="wmain"] /[type=="cell"][rowIdx in (1,-1)][colIdx in colRange(2,-2)][::groupRows()]>".

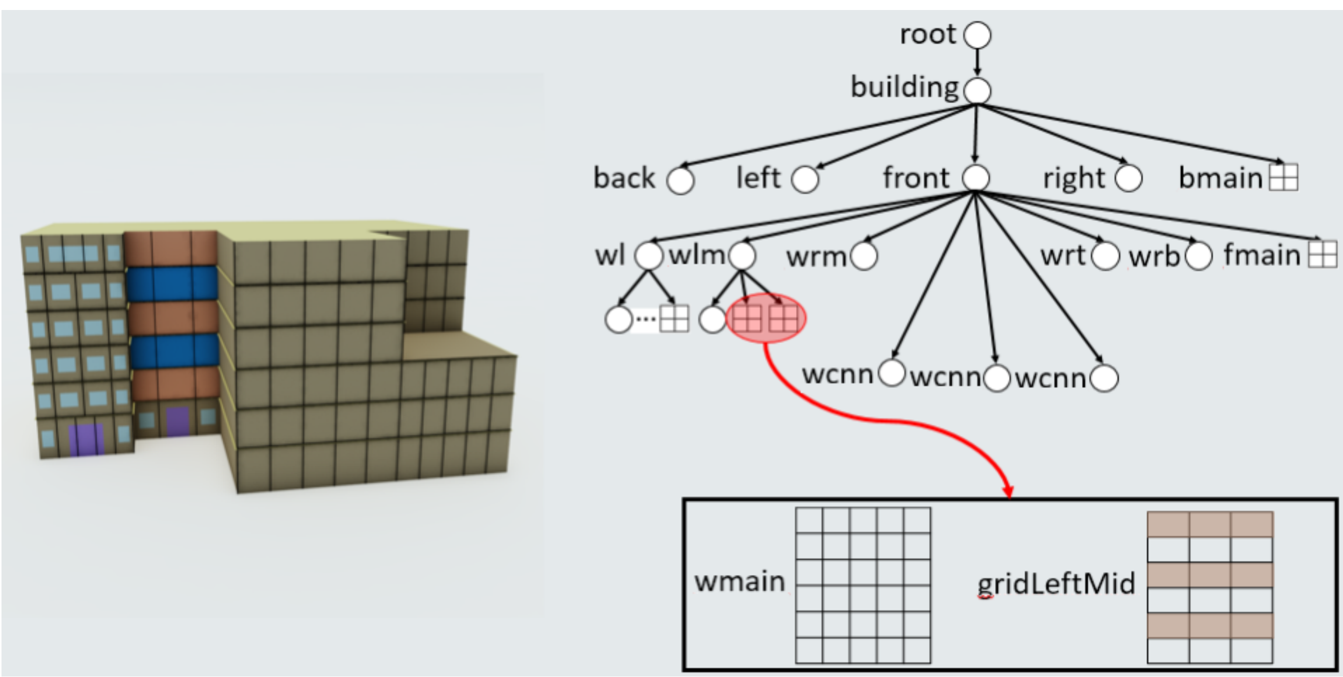


Figure 15: *(j) This selection selects grid cells in alternating floors to yield an alternating balcony pattern. The pattern selector* "pattern("(ab)*")=="a"" *selects every row with an odd number. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="wlm"] /[label=="gridLeftMid"] /[type=="cell"][rowIdx in rowRange(2,-1)][::groupRows()][pattern("(ab)*")=="a"]>".

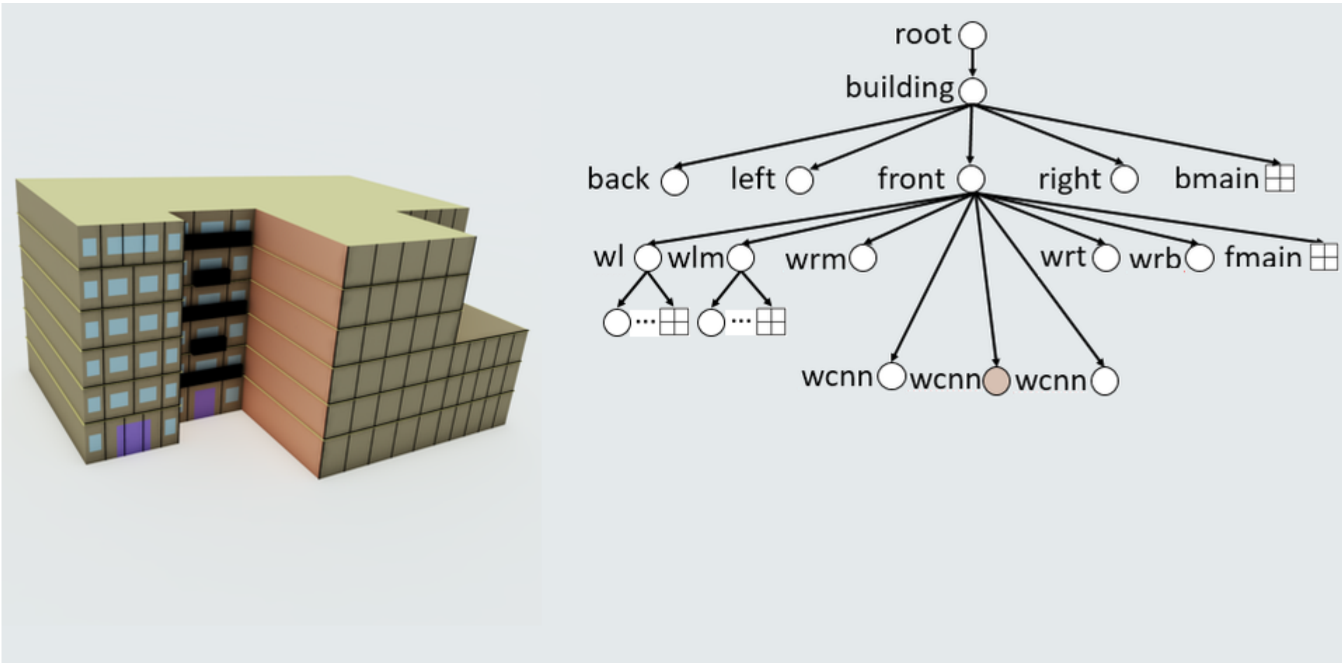


Figure 16: *(k) This selected facade is an example of geometry that emerged through an extrusion. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="wlm"] /neighbor("left", "right")>".

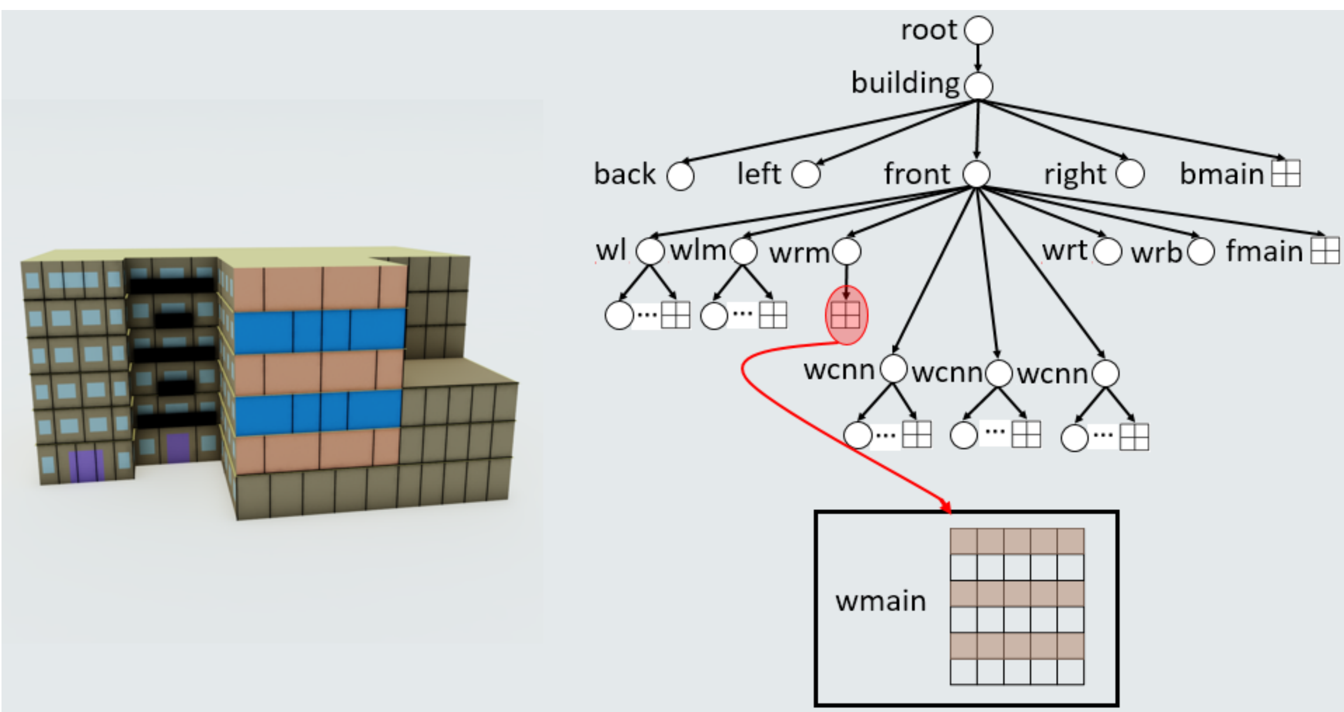


Figure 17: *(l) The corresponding selection-expression selects the second to the last floor in a subgrid and then selects every second floor. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="wrm"] /[label=="wmain"] /[type=="cell"][rowIdx in rowRange(2,-1)][::groupRows()][pattern("(ab)*") == "a"]>".

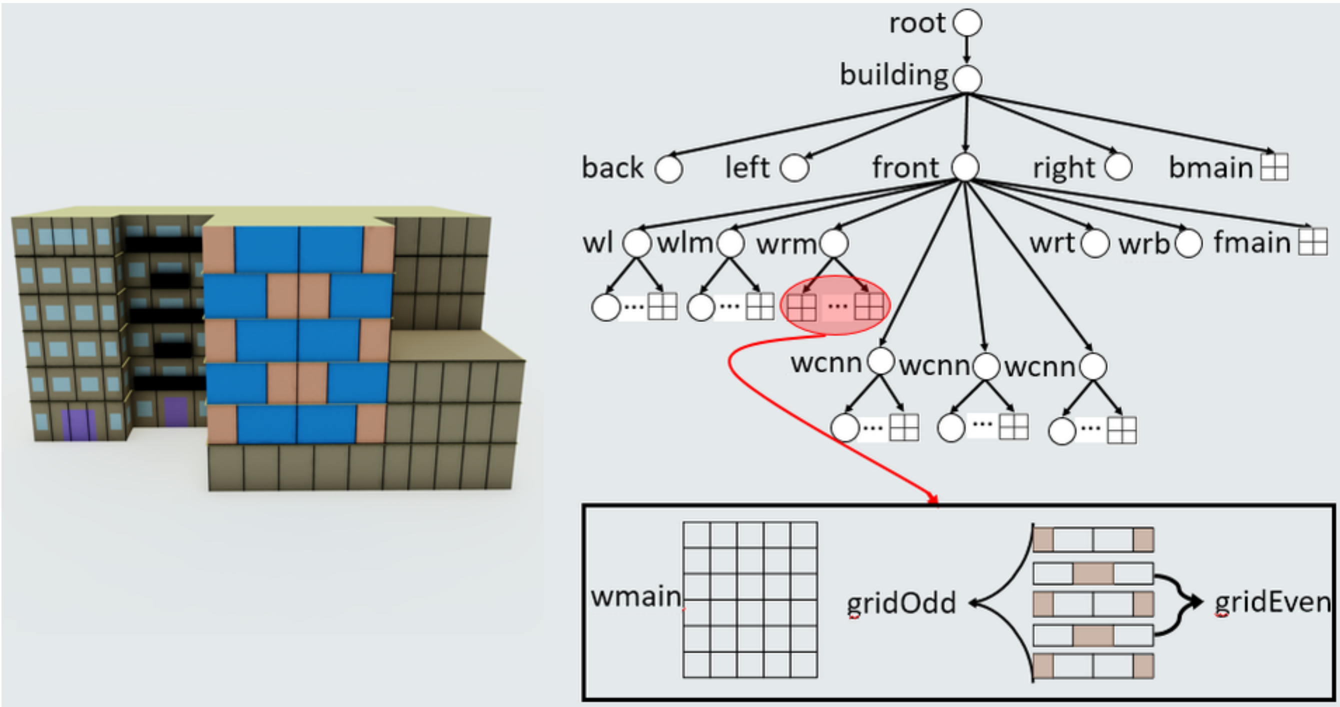


Figure 18: *(m) This complex selection pattern is done by referencing previously generated labels. The corresponding selection-expression is* "<descendant()[label=="front"] /[label=="wrm"] /[label in ("gridEven", "gridOdd")] /[type=="cell"][colLabel=="colNarrow"]>".

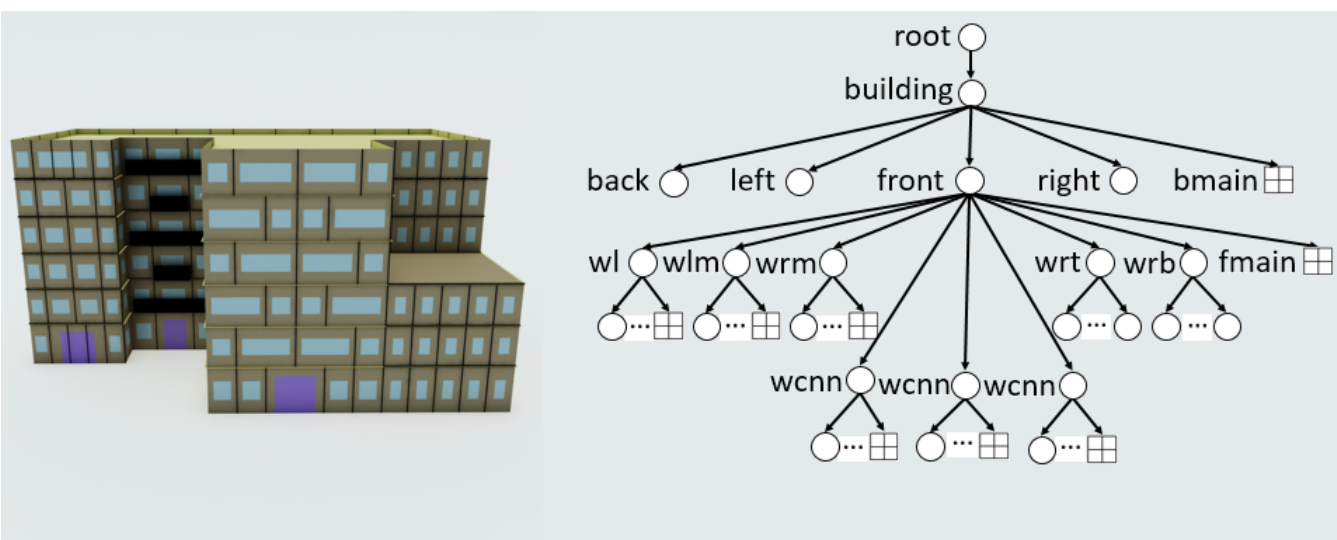


Figure 19: *(n) The final building without assets.*

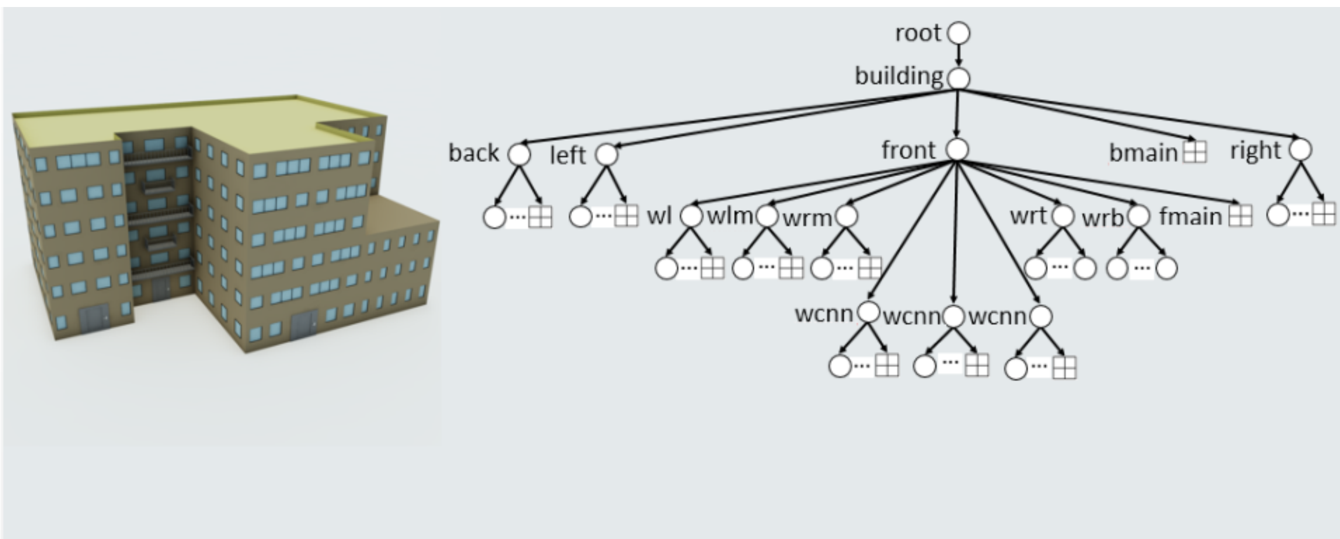


Figure 20: *(o) The final building including simpel assets for windows and doors.*